Chapter 1

Variant's generation

RQ1. To what extent it is possible to artificially create variants for WebAssembly programs?

In this chapter, we investigate whether we can artifically create program variants through semantically equivalent code transformations. We propose a framework to create program variants that are functionally equivalent to their original, yet exhibit a different behavior. We propose to retarget a superoptimizer, using its exhaustive searching strategy for providing semantically equivalent code transformations. We present a framework that is able to generate program variants that can be succesfully compiled to WebAssembly. The presented methodology and transformation tool, CROW, are contributions to this thesis. We evaluate the usage of CROW on a corpus of open-source and nature diverse programs. We sum up the key insights taken from this evaluation.

1.1 CROW

In this section we describe CROW, a tool tailored to create semantically equivalent variants out of a single program, either C/C++ code or LLVM bitcode. CROW is part of the contribution of this thesis. In Figure 1.1, we describe the workflow of CROW to create program variants.

CROW synthesizes program variants to be WebAssembly programs. We assume that the programs are generated through the LLVM compilation pipeline. This assumption is supported by the work of Lehman et al. [] and the fact that Wasm programs are generated in the 60% of the times by the LLVM toolchain.

During the *exploration* stage, CROW takes as input a C/C++ programs or LLVM bitcodes and produces a set of unique, diversified LLVM bitcodes that can be later ported to WebAssembly. Figure 1.1 shows the stages of this workflow. The workflow starts with compiling the input program into LLVM bitcode using clang

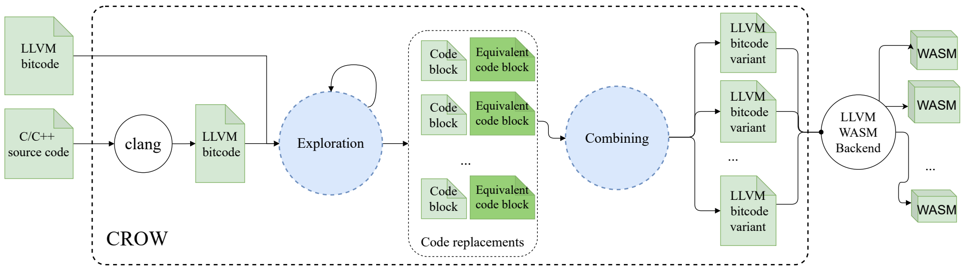


Figure 1.1: TODO

if it comes from source code or LLVM bicode. Then, CROW analyzes the bitcode to synthesize a set of candidate code replacements. In the following we enunciate the definitions we use along this work for code block, functional equivalence and code replacement.

Definition 1 *Block (based on Aho et al. [?]): Let P be a program. A block B is a grouping of declarations and statements in P inside a function F.*

Definition 2 *Functional equivalence modulo program state (based on Le et al. [?]): Let B_1 and B_2 be two code blocks according to Definition 1. We consider the program state before the execution of the block, S_i, as the input and the program state after the execution of the block, S_o, as the output. B_1 and B_2 are functionally equivalent if given the same input S_i both codes produce the same output S_o.*

Definition 3 *Code replacement: Let P be a program and T a pair of code blocks (B_1, B_2). T is a candidate code replacement if B_1 and B_2 are both functionally equivalent as defined in Definition 2. Applying T to P means replacing B_1 by B_2. The application of T to P produces a program variant P' which consequently is functionally equivalent to P.*

We address the *exploration* stage by retargeting a superoptimizer for LLVM, Souper, using its subset of the LLVM intermediate representation. CROW uses it at function level, taking the functions inside the LLVM bitcode module as individual instances to analyze and diversify/ The retargeted superoptimizer is in charge of finding the potential places in the original code functions where a replacement can be applied. Also, it makes the formal verification of Definition 2 using a theorem prover. On the other hand, we prevent the superoptimizer from synthesizing instructions that have no correspondence in WebAssembly for sake of reducing the searching space for equivalent program variants.

Finally, In the *combining* stage, CROW combines the candidate code replacements to generate different LLVM bitcode variants. In this stage, we select and combine code replacements that have been synthesized during the exploration stage. We apply each code replacement to the original program to produce a LLVM IR variant. Then, this IR is compiled into a WebAssembly binary if requested. CROW generates the variants from all possible combinations of code replacements as the power set over all code replacements.

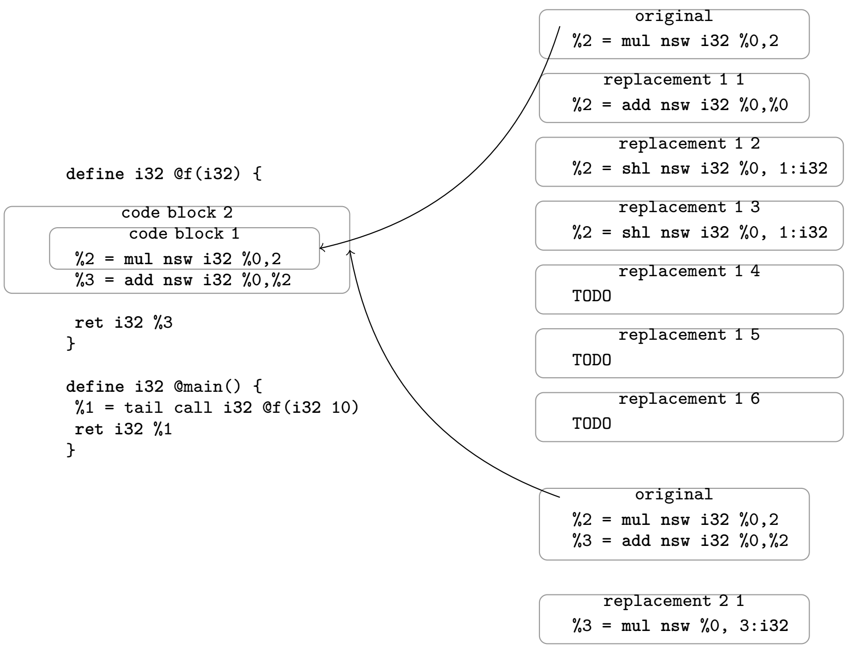
1.1.1 Example

Let us illustrate how CROW works with the simple example code in Listing 1.1. The `f` function calculates the value of $2*x+x$ where `x` is the input for the function. This code is compiled by CROW generating the intermediate LLVM bitcode in the left most part of Listing 1.2. As the right-most part of Listing 1.2 shows, CROW found two code blocks to look for variants.

Listing 1.1: C function that calculates the quantity $2x + x$

```
int f(int x) {
    return 2 * x + x;
}
```

Listing 1.2: LLVM's intermediate representation program, its extracted code blocks and replacement candidates.



```
define i32 @f(i32) {

code block 2
  code block 1
  %2 = mul nsw i32 %0,2
  %3 = add nsw i32 %0,%2

 ret i32 %3
}

define i32 @main() {
 %1 = tail call i32 @f(i32 10)
 ret i32 %1
}
```

```
original
%2 = mul nsw i32 %0,2

replacement 1 1
%2 = add nsw i32 %0,%0

replacement 1 2
%2 = shl nsw i32 %0, 1:i32

replacement 1 3
%2 = shl nsw i32 %0, 1:i32

replacement 1 4
TODO

replacement 1 5
TODO

replacement 1 6
TODO

original
%2 = mul nsw i32 %0,2
%3 = add nsw i32 %0,%2

replacement 2 1
%3 = mul nsw %0, 3:i32
```

TODO Add final Wasm here as examples

CROW, in the exploration stage, can synthesize $6 + 1$ bitcode variants for each code block respectively as the right most part of Listing 1.2 shows. The power set combination size of all possible combination of code replacements is 7×2 which results in 14 function variants. Yet, the generation stage would eventually generate 7 variants from the original WebAssembly binary. This gap between the number of potential and the actual number of variants is a consequence of the redundancy among the bitcode variants when composing several variants into one. Replacing the largest code block after the smallest will generate the same program. This can be appreciated in the last replacement candidate `%3 = mul nsw %0, 3:i32`, this replacement turns the code block2 into dead code, thus, no later replacement for it is applied.

1.2 Evaluation

We use CROW and method described at section 1.1 to answer RQ1. In this section we describe the corpora of original programs that we pass to CROW for sake of variants generation. Besides, we describe our metrics, and we finalize by discussing the results.

1.2.1 Corpora

We answer RQ1 with two corpora of programs appropriate for our experiments. The first corpus, **CROW prime**, is part of the CROW contribution []. The second corpus, **MEWE prime**, is part of the MEWE contribution []. In Table 1.1 we summarize the selection criteria, and we sum up the properties of each corpus. With both corpora we evaluate CROW wih a total of $303 + 7$ programs, containing 303+1902 functions. To assess the ability of our approach to generate WebAssembly binaries that are statically different, we compute the number of unique variants generated by CROW for each original program. We compare the WebAssembly program and its variant using the `md5` hash as a metric.

1.2.2 Setup and evaluation

CROW's workflow synthesizes program variants with an enumerative strategy. This means that all possible programs that can be generated for a given language (LLVM in the case) are constructed and verified for equivalence. There are two parameters to control the size of the search space and hence the time required to traverse it. On one hand, one can limit the size of the variants. On the other hand, one can limit the set of instructions that are used for the synthesis. In our experiments, we use between 1 instruction (only additions) and 60 instructions (all supported instructions in the synthesizer).

These two configuration parameters allow the user to find a trade-off between the amount of variants that are synthesized and the time taken to produce them. In

Corpus name	Selection criteria	Corpus Description
CROW prime	To build the corpus used in CROW, we take programs from the Rosetta Code project[1]. We first collect all C programs from Rosetta Code, which represents 989 programs as of 01/26/2020. We then apply a number of filters: the programs should successfully compile, they should not require user inputs, the programs should terminate and should not provide in non-deterministic results. The result of the filtering is a corpus of 303 C programs	All programs have a single function in terms of source code. These programs range from 7 to 150 lines of code and solve a variety of problems, from the *Babbage* problem to *Convex Hull* calculation.
MEWE prime	We select two mature and typical edge-cloud computing projects for this corpus. The projects are selected based on: suitability for diversity synthesis with CROW (the projects should have the ability to collect their modules in LLVM intermediate representation) The selected projects are: **libsodium**, an encryption, decryption, signature and password hashing library which can be ported to WebAssembly and **qrcode-rust**, a QrCode and MicroQrCode generator written in Rust. We then filter out 5 and 2 endpoints[2] respectively, for which we select their involved functions.	The evaluated projects contain in total 1902 functions, 62 for libdosium and 1840 for qrcode-rust. The function range between 10 ad 127700 lines of code.

Table 1.1: Corpora description. The table is composed by the name of the corpus, the selection criteria and the stats the programs in each corpus.

Table 1.2 we listed the configuration for both corpora. For the current evaluation, given the size of the corpus, we set the exploration time to 1 hour maximum per function for **CROW PRIME**, for a total of 303 hours CROW executions. In the case of **MEWE prime** we set the timeout to 5 minutes per function in the exploration stage. We set all 60 supported instructions in CROW for both **CROW prime** and **MEWE primer** corpora.

CORPUS	Exploration timeout	Max. instructions
CROW prime	1h	60
MEWE prime	5m	60

Table 1.2: CROW tweaking for variants generation. The table is composed by the name of the corpus, the timeout parameter and the maximum number of instructions allowed per variant.

1.3 Results

We summarize the results in Table 1.3 CROW produces at least one unique program variant for 239/303 programs for **CROW prime** with 1h for timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of **MEWE prime**, CROW produces equivalent variants for 48/1902 original programs with 5 minutes per function as timeout. The rest of the programs resulted timeout before finding function variants or produced none.

CORPUS	#Functions	# Diversified	# NonDiversified	# Variants
CROW prime	303	**239**	64	1906
MEWE prime	1902	48	1854	**4670**

Table 1.3: General diversification results. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the number of non-diversified functions and the number of unique variants.

TODO Expand this

1.3.1 Properties for large diversification using CROW

We made a manual analysis of the programs that yield more than 100 unique variants to study the key properties of programs leveraging a high number of variants. This reveals one key reason that favors many unique variants: the programs include bounded loops. In these cases CROW synthesizes variants for the loops by unrolling them. Every time a loop is unrolled, the loop body is copied and moved as part of the outer scope of the loop. This creates a new, statically different, program. The number of programs grows exponentially with nested loops.

A second key factor for the synthesis of many variants relates to the presence of arithmetic. Souper, the synthesis engine used by CROW, is effective in replacing arithmetic instructions by equivalent instructions that lead to the same result. For example, CROW generates unique variants by replacing multiplications with additions or shift left instructions (Listing 1.3). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 1.4). On the other hand, CROW is able to use the intrinsic of the computation model to create equivalent variants using overflow and underflow of integers to produce variants (Listing 1.5).

1.3.2 Challenges for automatic diversification

CROW generates variants for functions in both corpora, however, we have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions, as it can be appreciated in Table 1.3.

Listing 1.3: Diversification through arithmetic expression replacement.

```
local.get 0      local.get 0
i32.const 2      i32.const 1
i32.mul          i32.shl
```

Listing 1.4: Diversification through inversion of comparison operations.

```
local.get 0      i32.const 11
i32.const 10     local.get 0
i32.gt_s         i32.le_s
```

Listing 1.5: Diversification through overflow of integer operands.

```
i32.const 2      i32.const 2
i32.mul          i32.mul
                 i32.const -2147483647
                 i32.mul
```

CROW succesfully diversified approx. 79% and 2.5% of the original functions for **CROW prime** and **MEWE prime** respectively. On the other hand, CROW generated more variants for **MEWE prime**, 4670 program variants for 48 original programs. Not surprisingly, to set the timeout affects the capacity of CROW for diversification. On the other hand, a low timeout for exploration gives CROW more power of combining code replacements, generating more variants. This can be appreciated in the last column of the table, where for a lower number of diversified functions CROW created, overall, more variants.

Moreover, we look at the cases that yield a few variants per function. There is no direct correlation between the number of identified code for replacement and the number of unique variants. We manually analyze programs that include a significant number of potential places for replacements, for which CROW generates few variants. We identify three main challenges for diversification.

1) Constant computation We have observed that Souper searches for a constant replacement for more than 45% of the blocks of each program while constant values cannot be inferred. For instance, constant values cannot be inferred for memory load operations because CROW is oblivious to a memory model.

2) Combination computation The overlap between code blocks, mentioned in subsection 1.1.1, is a second factor that limits the number of unique variants. CROW can generate a high number of variants, but not all replacement combinations are necessarily unique. Let us illustrate the case with the example in Listing 1.2.

1.3.3 Variant properties

Regarding the potential size overhead of the generated variants, we have compared the WebAssembly binary size of the 239 programs with their variants. The ratio of size change between the original program and the variants ranges from 82% (variants are smaller) to 125% (variants are larger) for **CROW prime**. This limited impact on the binary size of the variants is good news because they are meant to save bandwidth when they become assets to distribute over network.

TODO Add the same for MEWE prime

1.4 Conclusions

The proposed methodology is able to generate program variants that are syntactically different to their original versions. We have shown that CROW generates diversity among the binary code variants. We identified the properties that original programs should have to provide many variants using CROW. Besides, we enumerated the challenges faced to provide automatic diversification.

In the next chapter we evaluate the assessment of the generated variants answering to what extent the artificially created variants are different from the original in terms of execution behavior.